Safely-Composable Type-Specific Languages

Cyrus Omar
Darya Kurilova
Ligia Nistor
Benjamin Chung
Alex Potanin *(Victoria University of Wellington)*
Jonathan Aldrich

School of Computer Science
Carnegie Mellon University

Specialized notations are important.

(mathematics)

There exists a positive constant M such that for all sufficiently large values of x, |f(x)| is at most M multiplied by x^2.

vs.

$$f(x) = \mathcal{O}(x^2)$$

Specialized notations are important.

(data structure literals)

```
Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
```

vs.

```
[1, 2, 3, 4, 5]
```

Specialized notations are important.

(regular expressions)

```
Concat(Digit, Concat(Digit, Concat(Char ':', Concat(Digit, Concat(Digit,
  Concat(ZeroOrMore(Whitespace), Group(Concat(Group(Or(Char 'a’, Char 'p')),
  Concat(Optional(Char '.'), Concat(Char 'm', Optional(Char '.'))))))))))))
```

vs.

```
/\d\d:\d\d\w?((a|p)\.?m\.?)/
```

Specialized notations are important.

(query languages like SQL)

```
exec_query(db, SelectQuery(["name", "ssn"], [
  WhereClause(EqualsPredicate("name”, StringLit(input)),
  FromClause("customers")]))
```

vs.

```
exec_query(db, <SELECT * FROM users WHERE name={name} AND pwhash={hash(pw)}>)
```

vs.

```
exec_query(db, "SELECT * FROM users WHERE name='"+name+"' AND pwhash='"+hash(pw)+"'")
```

```
"'; DROP TABLE users --"
```

Specialized notations are important.

(markup, layout, templating)

```
HTMLElement({}, [BodyElement({}, [H1Element({}, [TextNode “Results for ” + keyword])
,  ULElement({}, to_list_items(exec_query(db, SelectQuery([“title”, “snippet”], [
    WhereClause(InPredicate(StringLit(keyword), “title”)),
    FromClause(“results”)]))))]]]
```

vs.

```
<<html><body><h1>Results for {keyword}</h1><ul>
   {to_list_items(exec_query(db,
      <SELECT title, snippet WHERE {keyword} in title FROM results>))}
 </ul></body></html>>
```

vs.

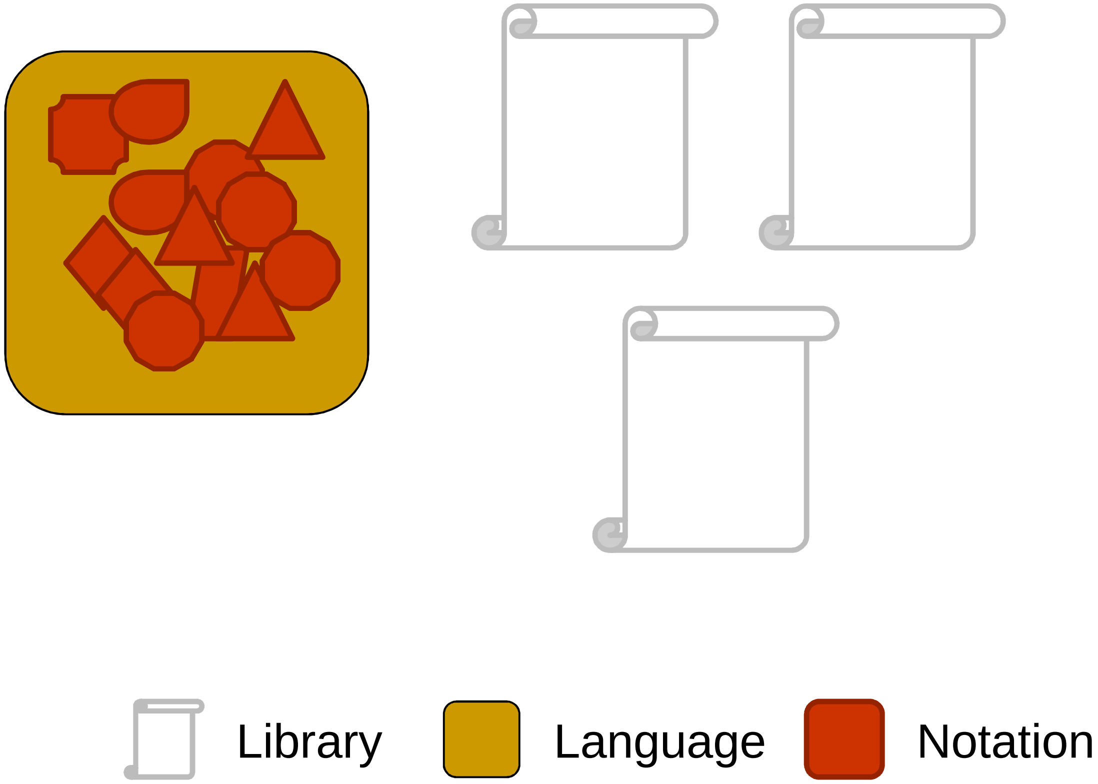
```
“<html><body><h1>Results for “ + keyword + “</h1><ul>” +
  to_list_items(exec_query(db,
    “SELECT title, snippet WHERE ‘” + keyword + “’ in title FROM results”)) +
“</ul></body></html>”
```

Wyvern

- **Goals:** Secure web and mobile programming within a single statically-typed language.

- Compile-time support for a variety of **domains**:
  - Security policies and architecture specifications
  - Client-side programming (HTML, CSS)
  - Server-side programming (Databases)

Monolithic languages where specialized notations must be anticipated and built in are unsustainable.

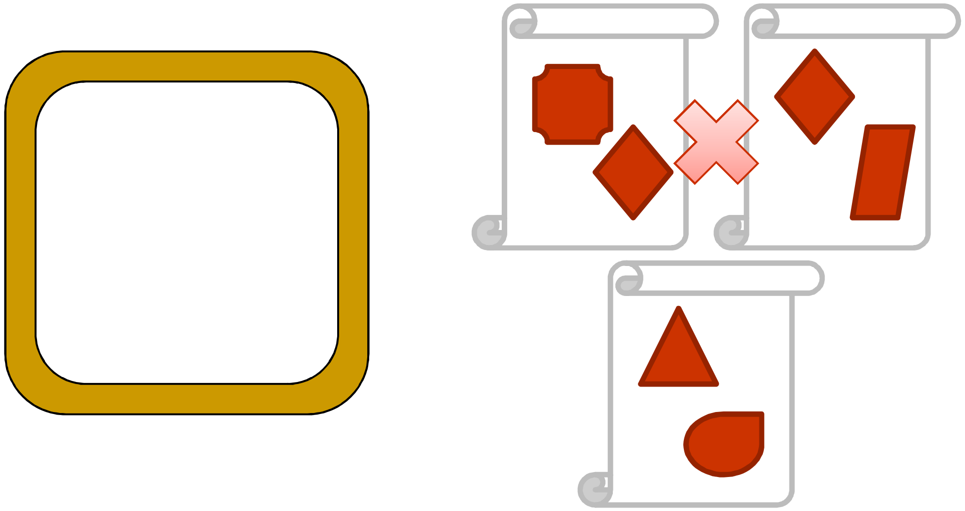


Library

Language

Notation

Better approach: an internally-extensible language where new notations can be distributed in libraries.



📜 Library 🟨 Language 🟥 Notation

Expressivity vs. Safety

- Want **expressive (syntax) extensions**.
- But if you give each DSL too much control, they may **interfere with one another** at link-time.

Example: Sugar* [Erdweg et al, 2010; 2013]

- Libraries can extend the **base syntax** of the language
- These extensions are imported **transitively**
- Extensions can **interfere**:
  - Pairs vs. Tuples
  - HTML vs. XML
  - Different implementations of the same syntax

Our Solution

- Libraries **cannot** extend the **base syntax** of the language
- Instead, **syntax is associated with types**.

“Type-specific languages” (TSLs)

- A type-specific language can be used within **delimiters** to **create values of that type**.

Examples: HTML and URLs

```
serve : (HTML, URL) -> unit
```

```
serve(~, `products.nameless.com`)
 html:
  head:
   title: Hot Products
   style: {myStylesheet}
   body:
    div id="search":
      {SearchBox("products")}
    div id="products":
      {FeedBox(servlet.hotProds())}
```

TSL Delimiters

- Several inline delimiters are available
  - `TSL code here, ``inner backticks`` must be doubled`
  - 'TSL code here, ''inner single quotes'' must be doubled'
  - {TSL code here, {inner braces} must be balanced}
  - [TSL code here, [inner brackets] must be balanced]
  - <TSL code here, <inner angle brackets> must be balanced>
  - others?

- If you use the tilde (~) with whitespace, there are no restrictions on the TSL code. **Layout determines the end of the block.**

Phase I: Top-Level Parsing

- The top-level layout-sensitive syntax of Wyvern can be parsed first without involving the typechecker
  - **Useful for tools like documentation generators**
  - **Wyvern's grammar can be written down declaratively using a layout-sensitive grammar formalism** [Erdweg et al. 2012; Adams 2013]

- TSL code (+Wyvern code inside it) is left **unparsed** during this phase.

Phase II: Typechecking and DSL Parsing

- When TSL code is encountered during typechecking, its **expected type** is determined via:
  - Explicit type annotations
  - Function signatures
  - Type propagation into **where** clauses

- The TSL is now parsed according to the **type-associated syntax**.
  - Any internal Wyvern expressions are also parsed (I & II) and typechecked recursively during this phase.

Associating a Parser with a type

```
casetype Regex =
  Digit | Char of char | Concat of Regex * Regex | …
  metaobject = new
    val parser : std.Parser = new
      def parse(s : TokenStream) : ExpAST =
         … code to parse regex literals …
```

```
type Parser =
  def parse(s : TokenStream) : AST
```

Associating a grammar with a type

```
casetype Regex =
  Digit | Char of char | Concat of Regex * Regex | …
  metaobject = new
    val parser : Parser = ~
      start ::= (“<“ t:tag “>” start “</” t:tag “>”)*
              | “{“ e:EXP “}”
        tag ::= ...
```

Benefits

- **Modularity and <u>Safe</u> Composability**
  - DSLs are distributed in libraries, along with types
  - No link-time errors
- **Identifiability**
  - Can easily see when a DSL is being used
  - Can determine which DSL is being used by identifying expected type
  - DSLs always generate a value of the corresponding type
- **Simplicity**
  - Single mechanism that can be described in a few sentences
  - Specify a grammar in a natural manner within the type
- **Flexibility**
  - Whitespace-delimited blocks can contain *arbitrary* syntax